

Echtzeitsysteme

Exkurs: WCET-Analyse

Lehrstuhl Informatik 4

24. Januar 2013

Gliederung

- 1 Überblick
- 2 Problemstellung
- 3 Pfadanalyse
 - Problemstellung
 - Timing Schema
 - Implicit Path Enumeration Technique
 - Übersicht
- 4 Hardware-Analyse
 - Cache-Analyse
- 5 Messbasierte WCET-Analyse
- 6 Zusammenfassung

Fragestellungen

- Alle sprechen von der **WCET** – aber wo kommt sie eigentlich her?
 - Wie geht man mit der Abhängigkeit von **Eingabedaten** um?
 - Welche Rolle spielt der **Prozessor** beim Thema WCET?
- **Pfadanalyse** \leadsto bestimmt den längsten Pfad durch ein Programm
 - **Timing Schema**
 - \leadsto orientiert sich an der Programmstruktur
 - **Implicit Path Enumeration Technique (IPET)**
 - \leadsto Abbildung auf ein **Flussproblem**
- **Hardware-Analyse** \leadsto bestimme Dauer des längsten Pfads
 - Prozessoren werden hinsichtlich **Geschwindigkeit** optimiert ...
 - Caches, Pipelines, Out-of-Order-Execution, Sprungvorhersage, ...
 - ... nicht hinsichtlich **Vorhersagbarkeit**
- Warum bestimmen wir die WCET nicht einfach durch **Messung**?

Gliederung

- 1 Überblick
- 2 Problemstellung
- 3 Pfadanalyse
 - Problemstellung
 - Timing Schema
 - Implicit Path Enumeration Technique
 - Übersicht
- 4 Hardware-Analyse
 - Cache-Analyse
- 5 Messbasierte WCET-Analyse
- 6 Zusammenfassung

Auf der Suche nach dem e

Die maximale Ausführungszeit ist eine unabkömmliche Information für die Ablaufplanung

Statische Ablaufplanung ordnet Jobs Zeitintervallen zu

↪ diese Zeitintervalle müssen ausreichend groß sein

- mindestens so groß, wie die maximale Ausführungszeit e des Jobs

Planbarkeitsanalyse basiert auf Abschätzungen ...

- der maximalen CPU-Auslastung:

$$\sum_{k=1}^n \frac{e_k}{\min(D_k, p_k)} + \frac{b_i^{np}}{\min(D_i, p_i)} \leq 1 \quad ; i = 1, 2, \dots, n$$

- der maximalen Antwortzeit:

$$\omega_i(t) = e_i + b_i^{np} + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k; 0 < t \leq p_i$$

die ohne die maximale Ausführungszeit e nicht auskommen

- selbst die Blockadezeit b^{np} ist eine maximale Ausführungszeit
- ↪ nämlich die des längsten kritischen Abschnitts

Warum ist es so schwierig, e zu bestimmen?

Anders: Wovon hängt die maximale Ausführungszeit eigentlich ab?

Beispiel: Bubblesort

```
void bubbleSort(int a[], int size) {
    int i, j;

    for(i = size - 1; i > 0; --i) {
        for(j = 0; j < i; ++j) {
            if(a[j] > a[j+1]) {
                swap(&a[j], &a[j+1]);
            }
        }
    }

    return;
}
```

Programmiersprachenebene:

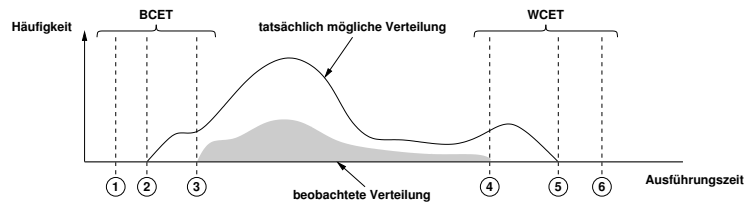
- Anzahl der Schleifendurchläufe hängt von der Größe des Feldes $a[]$ ab
- Anzahl der Vertauschungen (swap) hängt vom Inhalt des Feldes ab
- ↪ **exakte Vorhersage ist kaum möglich**
 - sowohl die Größe als auch der Inhalt des Feldes kann zur Laufzeit variieren

Die **Maschinenprogrammenebene** liefert Dauer der Elementaroperationen:

- wie lange dauert ein ADD, ein LOAD, ...
- ist **prozessorabhängig** und für moderne Prozessoren sehr schwierig
 - **Pipeline** ↪ Wie ist der Zustand der Pipeline an einer Instruktion?
 - **Cache** ↪ Liegt die Instruktion/das Datum im schnellen Cache?
 - **Out-of-Order-Execution, Branch-Prediction, Hyper-Threading, ...**

Aufgabenstellung

Die maximale Ausführungszeit nach oben abschätzen



Bereits für relativ einfache Programme ergibt sich eine Bandbreite möglicher Programmlaufzeiten, besondere Bedeutung haben

- bestmögliche Ausführungszeiten (engl. **best case execution time**)
 - ① geschätzt, ② tatsächlich und ③ beobachtet
- und maximale Ausführungszeiten (engl. **worst case execution time**)
 - ④ beobachtet, ⑤ tatsächlich und ⑥ geschätzt

📌 Ziel ist eine **sichere Abschätzung der WCET**

- und den Abstand zur tatsächlichen WCET klein zu halten

Gliederung

- 1 Überblick
- 2 Problemstellung
- 3 Pfdanalyse
 - Problemstellung
 - Timing Schema
 - Implicit Path Enumeration Technique
 - Übersicht
- 4 Hardware-Analyse
 - Cache-Analyse
- 5 Messbasierte WCET-Analyse
- 6 Zusammenfassung

Den längsten Weg durch ein Programm finden

Beispiel: Bubblesort

```
void bubbleSort(int a[],int size) {
    int i,j;
    for(i = size - 1; i > 0; --i) {
        for(j = 0; j < i; ++j) {
            if(a[j] > a[j+1]) {
                swap(&a[j],&a[j+1]);
            }
        }
    }
    return;
}
```

betrachte Aufrufe: bubbleSort(a, s)

- Anzahl von **Durchläufen**, **Vergleichen** und **Vertauschungen** (engl. **Swap**)
- $a = \{1, 2\}, s = 2$
 $\sim D = 1, V = 1, S = 0;$
- $a = \{1, 3, 2\}, s = 3$
 $\sim D = 3, V = 3, S = 1;$
- $a = \{3, 2, 1\}, s = 3$
 $\sim D = 3, V = 3, S = 3;$

- ist für den **allgemeinen Fall nicht berechenbar** \sim **Halteproblem**

- Wieviele Schleifendurchläufe werden benötigt?

\sim in Echtzeitsystemen ist dieses Problem aber häufig lösbar

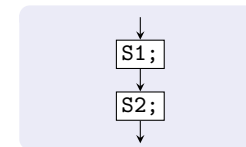
- **kanonische Schleifenkonstrukte**: `for(int i = 0; i < X; ++i)`
 - X ist oft eine Konstante oder zumindest beschränkt
 - ggf. muss die obere Schranke manuell annotiert werden
- die **maximale**, nicht die **exakte Pfadlänge** ist von Belang

Abstrakter Syntaxbaum \sim Timing Schema

Ableitung des maximalen Pfads anhand der Programmstruktur

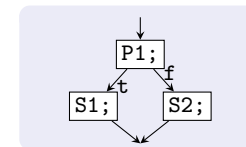
Sequenzen \sim Hintereinanderausführung

Summation der WCETs:
 $e_{seq} = e_{S1} + e_{S2}$



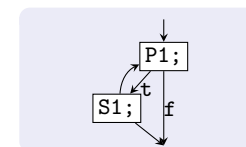
Verzweigung \sim bedingte Ausführung

Abschätzung der Gesamtausführungszeit:
 $e_{cond} = e_{P1} + \max(e_{S1}, e_{S2})$



Schleifen \sim wiederholte Ausführung

Schleifendurchläufe berücksichtigen:
 $e_{loop} = e_{P1} + n(e_{P1} + e_{S1})$



Beispiel: Bubblesort

```
void bubbleSort(int a[],int size) {
    int i,j;
    for(i = size - 1; i > 0; --i) {
        for(j = 0; j < i; ++j) {
            if(a[j] > a[j+1]) {
                swap(&a[j],&a[j+1]);
            }
        }
    }
    return;
}
```

- Schleife $L_1: P_1 = i > 0$
 - Rumpf: $L_2; --i;$
 - Durchläufe: $size - 1$
 $\sim e_{L_1} = e_{P_1} + (size - 1)(e_{P_1} + e_{L_2} + e_{--i})$
- Schleife $L_2: P_2 = j < i$
 - Rumpf: $C_1; ++j;$
 - Durchläufe: $size - 1$
 $\sim e_{L_2} = e_{P_2} + (size - 1)(e_{P_2} + e_{C_1} + e_{++j})$

- Verzweigung $C_1: P_3 = a[j] > a[j + 1]$

- $S_1 = swap(&a[j], &a[j + 1])$

$\sim e_{C_1} = e_{P_3} + e_{S_1}$

- Funktionsaufruf $S_1 = swap(&a[j], &a[j + 1])$

- analog zum hier vorgestellten Verfahren

Timing Schema: Vor- und Nachteile

- Traversierung des abstrakten Syntaxbaums *bottom-up*
 - d. h. an den Blättern beginnend, bis man zur Wurzel gelangt
- maximale Ausführungszeit wird nach festen Regeln akkumuliert
 - für Sequenzen, Verzweigungen und Schleifen

Vorteile

- + einfaches Verfahren mit geringem Berechnungsaufwand
- + skaliert gut mit der Programmgröße

Nachteile

- generische Flussinformation kann kaum berücksichtigt werden
 - z. B. sich ausschließende Zweige aufeinanderfolgender Verzweigungen
- schwierige Integration mit einer separaten Hardware-Analyse

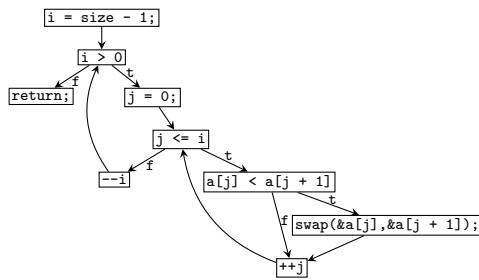
Den längsten Weg durch ein Programm finden

Die möglichen Wege lassen sich durch Kontrollflussgraphen beschreiben

Ein **Kontrollflussgraphen** (engl. *control flow graph*) ist ein gerichteter Graph und setzt sich aus **Grundblöcken** (engl. *basic blocks*) zusammen

- Grundblöcke sind sequentielle „Code-Schnipsel“
 - hier wird gearbeitet \rightsquigarrow Grundblöcke verbrauchen Rechenzeit
- Kanten im Kontrollflussgraphen \rightsquigarrow Sprünge zwischen Grundblöcken

```
void bubbleSort(int a[],int size) {
    int i,j;
    for(i = size - 1; i > 0; --i) {
        for(j = 0; j < i; ++j) {
            if(a[j] > a[j+1]) {
                swap(&a[j],&a[j+1]);
            }
        }
    }
    return;
}
```



Wie berechnet man den längsten Pfad?

... und bestimmt so auch gleich die maximale Ausführungszeit

Lösungsansatz: Fasse die Bestimmung der WCET als Flussproblem auf [2]

- die **Maximierung des Flusses** in einem gerichteten Graphen ist ein gut untersuchtes Problem aus dem Bereich der Graphentheorie
- mithilfe **ganzzahliger linearer Programmierung** (engl. *integer linear programming*) lässt sich dieses Problem zudem effektiv lösen

Vorgehen: Transformiere den Kontrollflussgraphen in ein ganzzahliges, lineares Optimierungsproblem (ILP) und löse es

- 1 bestimme einen **Zeitanalysegraph** aus dem Kontrollflussgraphen
- 2 formuliere das lineare Optimierungsproblem
- 3 bestimme die **Flussrestriktionen** des Zeitanalysegraphen
 - dies sind die Nebenbedingungen im Optimierungsproblem
- 4 löse des Optimierungsproblem (z.B. mit *Ipsolve*¹)

¹<http://lpsolve.sourceforge.net/>

Der Zeitanalysegraph (engl. *timing analysis graph*)

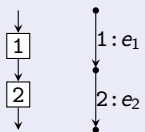
Ein **Zeitanalysegraph** (oder kurz **T-Graph**) ist ein gerichteter Graph mit einer Menge von Knoten $\mathcal{V} = \{V_i\}$ und Kanten $\mathcal{E} = \{E_i\}$.

- mit genau einer **Quelle** und einer **Senke**
 - Knoten, aus denen/in die nur Kanten entspringen/münden
- jede Kante ist Bestandteile eines Pfads P von der Senke zur Kante
 - solche ein Pfad P entspricht einer möglichen Abarbeitung
- jeder Kante wird ihre WCET e_i zugeordnet

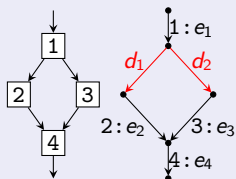
Grundblöcke des Kontrollflussgraphen werden auf Kanten abgebildet

- für Verzweigungen benötigt man **Dummy-Kanten d_i**

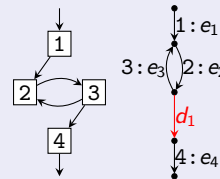
Sequenz



Verzweigung



Schleife



Bestimmung der WCET

Mit der Anzahl f_i der Ausführungen einer Kante E_i bestimmt man die WCET e durch Summation der Ausführungszeiten des längsten Pfades:

$$e = \max_P \sum_{E_i \in P} f_i e_i$$

Problem: erfordert die **explizite Aufzählung aller Pfade**

\rightsquigarrow das ist algorithmisch nicht handhabbar

Lösung: fasse die Bestimmung der WCET als **Flussproblem** auf

\rightsquigarrow der **maximale Fluss** durch das durch den T-Graphen gegebene Netzwerk führt zur gesuchten WCET

\rightsquigarrow Flussprobleme sind mathematisch gut untersucht und lassen sich durch **lineare Ganzzahlprogrammierung** lösen

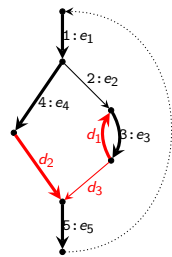
Zirkulationen

Eine Abbildung $f : \mathcal{E} \mapsto \mathcal{R}$ heißt **Zirkulation**, falls sie den Fluss erhält

- jeder Kante wird die **Zahl der Ausführungen** f_i als Fluss zugeordnet
- **Flusserhaltung**: jeder Knoten wird gleich oft betreten und verlassen
 - erfordert die Einführung einer Rückkehrkante E_e mit $f_e = 1$

Flussrestriktionen schließen Zirkulationen ungültiger Abarbeitungen aus

- Formulierung als **Nebenbedingungen** des Optimierungsproblems
- Beschränkung der maximalen Anzahl von Schleifendurchläufen



- $f_1 = f_2 + f_4$ wird durch die Zirkulation garantiert
- gültige Zirkulation: $\{E_1, E_4, d_2, E_5, E_e\} \cup \{E_3, d_1\}$
 ~ aber **keine gültige Abarbeitung**
- Flussrestriktion $f_3 \leq 5f_2$ löst dieses Problem
 - wird E_2 nicht abgearbeitet, so gilt $f_3 \leq 5 \cdot 0 = 0$
 - hier: Beschränkung auf 5 Schleifendurchläufe
 - ~ Nebenbedingung des Optimierungsproblems

Ganzzahliges Lineares Optimierungsproblem

Zielfunktion: Maximierung des gewichteten Flusses

$$WCET_e = \max_{(f_1, \dots, f_e)} \sum_{E_i \in \mathcal{E}} f_i e_i$$

~ der Vektor (f_1, \dots, f_e) maximiert die Ausführungszeit

Nebenbedingungen garantieren tatsächlich mögliche Ausführungen

- **Flusserhaltung** für jeden Knoten des T-Graphen

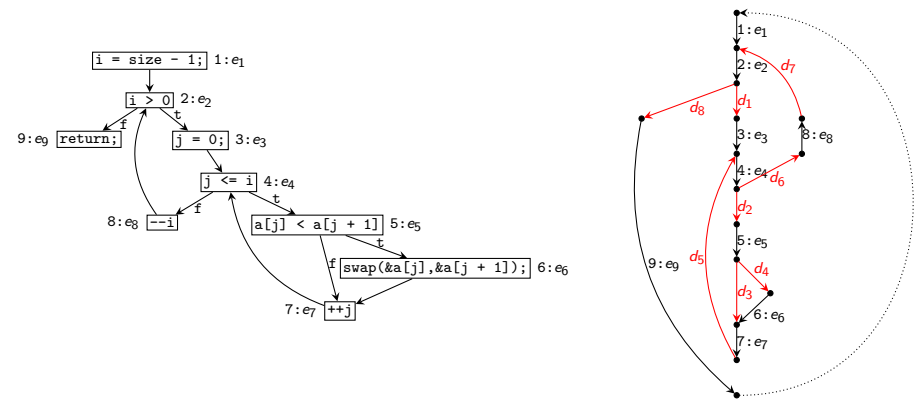
$$\sum_{E_j^+ = V_i} f_j = \sum_{E_k^- = V_i} f_k$$

- **Flussrestriktionen** für alle Schleifen des T-Graphen, z.B.

$$f_2 \leq (size - 1)f_1$$

- **Rückkehrkante** kann nur einmal durchlaufen werden: $f_{E_e} = 1$

Beispiel: Bubblesort



- Flussrestriktionen, die sich aus Schleifen ergeben:
 - „äußere Schleife“: $f_2 \leq (size - 1)f_1$
 - „innerer Schleife“: $f_4 \leq (size - 1)f_3$
- Flussrestriktionen, die sich aus Verzweigungen ergeben:
 - bedingte Vertauschung: $f_{d_3} + f_6 = f_7$

Vor- und Nachteile

- betrachte mögliche Abarbeitungen des Kontrollflussgraphen
- dabei werden alle Pfade implizit in Betracht gezogen
 - zunächst wird aus dem Kontrollflussgraph ein T-Graph erzeugt
 - dieser wird in ganzzahliges lineares Optimierungsproblem überführt

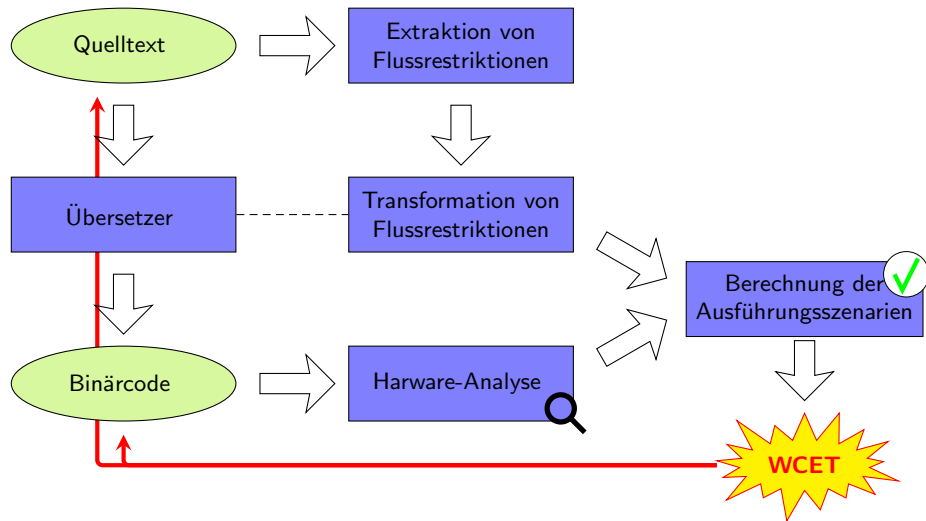
Vorteile

- + Möglichkeit komplexer Flussrestriktionen
 - z. B. sich ausschließende Äste aufeinanderfolgender Verzweigungen
- + Nebenbedingungen für das ILP sind leicht aufzustellen
- + viele Werkzeuge zur Lösung von ILPs verfügbar

Nachteile

- das Lösen eines ILP ist im Allgemeinen **NP-hart**
- auch Flussrestriktionen sind kein Allheilmittel
 - Beschreibung der Ausführungsreihenfolge ist problematisch

Werkzeugkette für die WCET-Analyse [3]



Gliederung

- 1 Überblick
- 2 Problemstellung
- 3 Pfadanalyse
 - Problemstellung
 - Timing Schema
 - Implicit Path Enumeration Technique
 - Übersicht
- 4 Hardware-Analyse
 - Cache-Analyse
- 5 Messbasierte WCET-Analyse
- 6 Zusammenfassung

Wie lange dauern die „sequentiellen Code-Schnipsel“

Die WCETs e_i der einzelnen Grundblöcke ist Eingabe für die Flussanalyse

Grundproblem: Ausführungszyklen von Instruktionen zählen

```
-getop:
link    a6,#0      ; 16 Zyklen
moveml  #0x3020,sp@- ; 32 Zyklen
move1   a6@(8),a2  ; 16 Zyklen
move1   a6@(12),d3 ; 16 Zyklen
```

Quelle: Peter Puschner [2]

- Ergebnis: $e_{\text{getopt}} = 80$ Zyklen
- Annahmen:
 - obere Schranke für jede Instruktion
 - die obere Schranke der Sequenz bestimmt man durch Summation

Problem: Vorgehen ist **äußerst pessimistisch** und **zum Teil falsch**

falsch für Prozessoren mit **Laufzeitanomalien**

- WCET der Sequenz > Summe der WCETs aller Instruktionen

pessimistisch für **moderne Prozessoren**

- Pipeline, Cache, Branch Prediction, Prefetching, ... haben großen Anteil an der verfügbaren Rechenleistung heutiger Prozessoren
- blanke Summation einzelner WCETs ignoriert diese Maßnahmen

Hardware-Analyse

Hardware-Analyse teilt sich in verschiedene Phasen

- Aufteilung ist nicht dogmenhaft festgeschrieben

Integration von Pfad- und Cache-Analyse

- 1 Pipeline-Analyse
 - Wie lange dauert die Ausführung der Instruktionssequenz?
- 2 Cache- und Pfad-Analyse sowie WCET-Berechnung
 - Cache-Analyse wird direkt in das Optimierungsproblem integriert

Separate Pfad- und Cache-Analyse

- 1 Cache-Analyse
 - kategorisiert Speicherzugriffe mit Hilfe einer Datenflussanalyse
- 2 Pipeline-Analyse
 - Ergebnisse der Cache-Analyse werden direkt berücksichtigt
- 3 Pfad-Analyse und WCET-Berechnung

Cache-Analyse [4, Kapitel 22]

Cache: ein kleiner, schneller Zwischenspeicher, Zugriffszeiten auf Daten/Instruktionen variieren je nach Zustand des Caches enorm:

Treffer (engl. *hit*), Daten/Instruktion sind im Cache $\rightsquigarrow e_h$

Fehlschlag (engl. *miss*), Daten/Instruktion sind nicht im Cache $\rightsquigarrow e_m$

Hits sind schneller als Misses: $e_m \gg e_h$ (> 100 Taktzyklen möglich)

Folgende Eigenschaften von Caches haben Einfluss auf seine Analyse

- Typ**
- Cache für **Instruktionen**
 - Cache für **Daten**
 - **kombinierter** Cache für **Instruktionen und Daten**
- Auslegung**
- **direkt abgebildet** (engl. *direct mapped*)
 - **vollasoziativ** (engl. *fully associative*)
 - **satz- oder mengenassoziativ** (engl. *set associative*)

Seitenersetzungsstrategie

- engl. *(pseudo) least recently used*, (Pseudo-)LRU
- engl. *(pseudo) first in first out*, (Pseudo-)FIFO

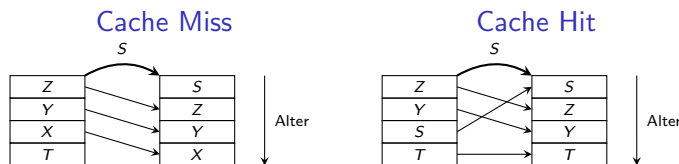
Beispiel: LRU-Cache, 4-fach assoziativ

LRU = „least recently used“ – Das älteste Element fliegt raus!

Caches werden häufig in **Sätze** (engl. *cache set*) unterteilt

- ein **n-fach assoziativer Cache** besitzt pro Satz **n** Cache-Blöcke
 - \rightsquigarrow Aufnahme von **n** konkurrierende Speicherstellen pro Satz möglich
- Inhalt und Verwaltungsinformation (bei LRU das Alter des Blocks) werden sowohl bei Treffern als auch bei Fehlschlägen aktualisiert

konkrete Semantik des Cache



must-Analyse und may-Analyse approximieren diese konkrete Semantik:

must Obergrenze des Alters \rightsquigarrow Unterapproximation des Inhalts

- Obergrenze \leq Assoziativität \rightsquigarrow garantiert im Cache

may Untergrenze des Alters \rightsquigarrow Überapproximation des Inhalts

- Untergrenze $>$ Assoziativität \rightsquigarrow garantiert nicht im Cache

Ergebnisse der Cache-Analyse

Hilfreich ist, zu wissen, ob z.B. eine Instruktion im Cache ist, oder nicht:

must, die Instruktion ist **garantiert im Cache**

- \rightsquigarrow man kann immer die schnellere Ausführungszeit e_h annehmen
- wird für die Vorhersage von Treffern verwendet

may, die Instruktion ist **vielleicht im Cache**

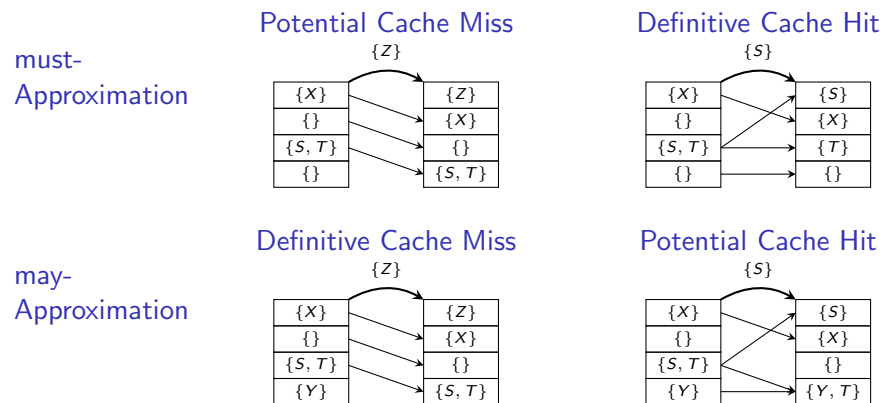
- \rightsquigarrow ist dies nicht der Fall, muss man die Ausführungszeit e_m annehmen
- wird für die Vorhersage von Fehlschlägen verwendet

persistent, die Instruktion **verbleibt im Cache**

- \rightsquigarrow erster Zugriff ist ein Fehlschlag, alle weiteren sind Treffer
- \rightsquigarrow erster Zugriff: e_m , weitere Zugriffe: e_h
 - ist besonders für Schleifen interessant, die den Cache „füllen“

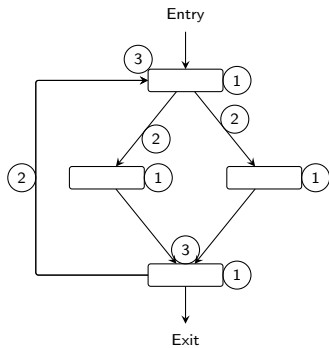
Beispiel: LRU-Cache, Zugriff auf eine Speicherstelle

Annäherung des Cache-Verhaltens durch must- und may-Approximation:
Aktualisierung von Inhalt und Verwaltungsinformation



Wie funktioniert nun die Cache-Analyse?

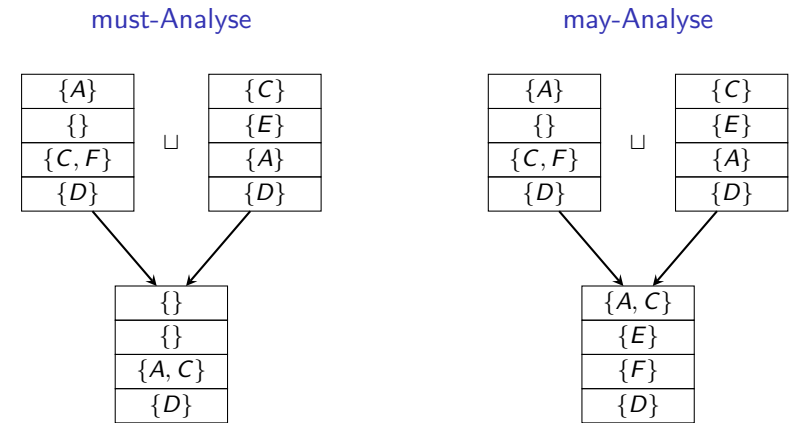
Die Analyse ist eine **Datenflussanalysen** [1, Kapitel 8]



- 1 sammle Information in den Grundblöcken
 - Speicherzugriffe (s. Folie IX/28)
 - man bestimmt die **Übertragungsfunktion** (engl. *transfer function*) des Grundblocks
- 2 die Information wird über ausgehende Kanten weiterverteilt
 - Eingabe für die Übertragungsfunktion der folgenden Grundblöcke
- 3 fließt der Kontrollfluss wieder zusammen, wird auch die Information verschmolzen
 - ~ Verschmelzungsoperatoren

Verschmelzungsoperatoren für must- und may-Analyse

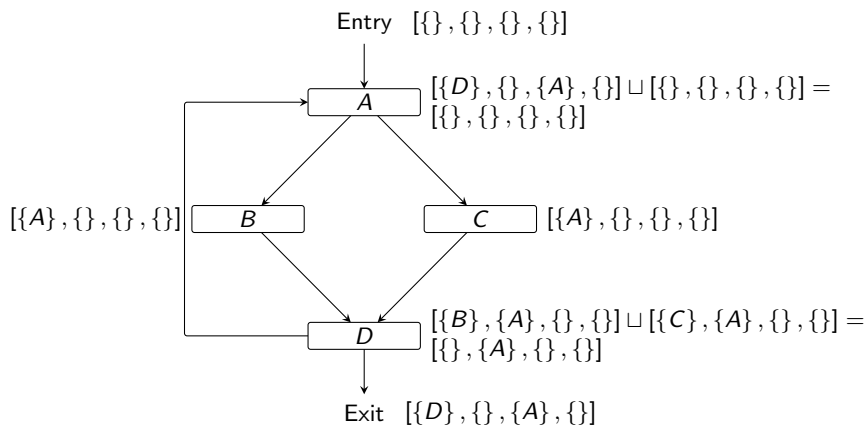
Verschmelzungsoperatoren für must- und may-Analyse



„Schnittmenge + max. Alter“

„Vereinigungsmenge + min. Alter“

Beispiel: must-Analyse für LRU



Hier ist leider keine Vorhersage von Treffern möglich ☹
 ~ Tip: ein einfaches, virtuelles **Ausrollen** der Schleife hilft weiter!

Praxisrelevante Cache-Implementierungen

Cache-Analyse mithilfe einer Datenflussanalyse funktioniert für **mengenassoziative Caches mit LRU** sehr gut

- Zugriffe auf unterschiedliche Cache-Zeilen beeinflussen sich nicht
- TriCore: 2-fach assoziativer LRU-Cache

Zur Leistungssteigerung kommen auch andere Strategien zum Einsatz:

- im Durchschnitt ähnliche Leistung wie LRU, **weniger vorhersagbar**
- **Pseudo-LRU**
 - Cache-Zeilen werden als Blätter eines Baums verwaltet
 - must-Analyse **ingeschränkt brauchbar**, may-Analyse **unbrauchbar**
 - z. B. PowerPC 750/755
- **Pseudo-Round-Robin**
 - 4-fach mengenassoziativer Cache mit **einem** 2-bit Ersetzungszähler
 - Zähler deutet auf zu ersetzende Cache-Zeile, Erhöhung bei Fehlschlag
 - must-Analyse **kaum**, may-Analyse **überhaupt nicht brauchbar**
 - z. B. Motorola Coldfire 5307

Gliederung

- 1 Überblick
- 2 Problemstellung
- 3 Pfadanalyse
 - Problemstellung
 - Timing Schema
 - Implicit Path Enumeration Technique
 - Übersicht
- 4 Hardware-Analyse
 - Cache-Analyse
- 5 **Messbasierte WCET-Analyse**
- 6 Zusammenfassung

Messbasierte WCET-Analyse [3]

Idee: der Prozessor selbst ist das präziseste Hardware-Modell

~> Führe das Programm aus und beobachte die Ausführungszeit!

Probleme messbasierter Ansätze

- in der Praxis ist es unmöglich **alle relevanten Pfade** zu betrachten
- **gewählte Testdaten** führen nicht unbedingt zum **längsten Pfad**
- **seltene Ausführungsszenarien** werden nicht abgedeckt
- **abschnittsweise WCET-Messung** ↗ globalen WCET
- schwierig/unmöglich den **Startzustand des Prozessors** zu identifizieren/erzwingen, der zur WCET führt

👉 messbasierte Ansätze unterschätzen die WCET meistens

👉 systematischere, messbasierte Analysetechniken sind vonnöten

Messbasierte WCET-Analyse [3] (Forts.)

Andererseits besteht Bedarf für messbasierte Methoden

- **gängige Praxis** in der Industrie
- nicht alle Echtzeitsysteme benötigen eine sichere WCET
 - z. B. Echtzeitsystem mit **weichen Zeitschranken** (engl. *soft real-time systems*)
- lassen sich leicht an **neue Hardwareplattformen** anpassen
 - häufig ist kein geeignetes statisches Analysewerkzeug verfügbar
- **geringer Aufwand für Annotationen**
 - verschafft leicht Orientierung über die tatsächliche Laufzeit
- **sinnvolle Ergänzung** zur statischen WCET-Analyse
 - **Validierung** statisch bestimmter Werte
 - Ausgangspunkt für die Verbesserung der statischen Analyse

👉 **Allerdings** sollte man nicht „einfach draus los messen“

~> z. B. immer Pfade vermessen (d. h. Ablauf und Zeit)

~> auf einen definierten Startzustand achten

Gliederung

- 1 Überblick
- 2 Problemstellung
- 3 Pfadanalyse
 - Problemstellung
 - Timing Schema
 - Implicit Path Enumeration Technique
 - Übersicht
- 4 Hardware-Analyse
 - Cache-Analyse
- 5 **Messbasierte WCET-Analyse**
- 6 **Zusammenfassung**

Resümee

Problemdefinition identifiziert zwei Teilprobleme

- **Flussanalyse** finde die längsten Pfade durch ein Programm
- **Hardwareanalyse** bestimmt die WCET einzelner Grundblöcke

Pfadanalyse \leadsto findet auf Programmiersprachenebene statt

- **Timing Schema** verwendet den abstrakten Syntaxbaum
- **IPET** basiert auf dem Kontrollflussgraphen
 - Erzeugung eines **T-Graphen**, Ableitung eines **Flussproblems**
 - \leadsto ganzzahliges lineares Optimierungsproblem

Werkzeugkette für die statische WCET-Analyse

Hardware-Analyse \leadsto Eingaben für die WCET-Berechnung

- Hauptaufgaben: **Cache-** und **Pipeline-Analyse**
- Beispiel: Datenflussanalyse für 4-fach assoziativen LRU-Cache
 - must-Approximation und may-Approximation

messbasierte WCET-Analyse \leadsto ein kleiner Fingerzeig!

Literaturverzeichnis

- [1] MUCHNICK, S. S.:
Advanced compiler design and implementation.
San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1997. – ISBN 1-55860-320-4
- [2] PUSCHNER, P. :
Zeitanalyse von Echtzeitprogrammen.
Treitlstr. 1-3/182-1, 1040 Vienna, Austria, Technische Universität Wien, Institut für Technische Informatik, Diss., 1993
- [3] PUSCHNER, P. ; HUBER, B. :
Zeitanalyse von sicherheitskritischen Echtzeitsystemen.
<http://ti.tuwien.ac.at/rts/teaching/courses/wcet>, 2012. – Lecture Notes
- [4] WILHELM, R. :
Embedded Systems.
<http://react.cs.uni-sb.de/teaching/embedded-systems-10-11/lecture-notes.html>, 2010. – Lecture Notes